Slide additions

This document contains a couple of topics that the book did not cover.

Trace scheduling

Trace scheduling is a Static scheduling technique that uses advanced compiler algorithms and profiling techniques to identify the most commonly executed *traces*—execution paths—in a program and then optimise their execution to increase CPI.

This technique is normally applied to VLIW processor architectures.

Since trace scheduling cannot proceed beyond loop barriers, traces never include loops. In order to try and mitigate this issue, some loop unrolling techniques may be used. However, loop unrolling increases the code size and the overhead needed to execute multiple iterations of the loop, so it has some performance drawbacks.

Of course, trace scheduling "bets" on which trace is executed more, which means that when program order takes the execution to a branch not optimised for, some compensation must happen. This is generally of two kinds:

1. Moving an instruction to a **side exit**: when the initial part of a trace is executed but not the rest, the execution can be detoured to the correct code fairly easily.
2. Moving an instruction to a **side entry**: when the final part of a trace is executed but not its beginning, the execution can be brought from the "excluded" code to the expected trace code; however, this is a costly operation.

Unfortunately, code cannot move freely in a trace, since the dataflow of the program must remain the same and exception behaviour must be maintained. Keeping data and control dependencies as in program order is key to achieve these two properties.

Furthermore, control dependencies can be eliminated via:

- Speculative scheduling
- Hyperblock scheduling (or predicated execution)

Predicated execution essentially eliminates branches by decomposing branches into sequential instructions accompanied by a predicate, which changes what instruction is executed according to its value.

Rotating register files

Rotating register files reduce the need for a high number of registers by allocating a new set of registers for each loop iteration.

By using rotating registers, loop iterations can use the same data when needed and avoid conflicts when they occur: while the logical register for different instructions in different iterations of a loop may remain the same, the rotating register file "maps" them to a different physical register, providing a sort of register renaming scheme without the need for a large number of register files.

Software pipelining

Software pipelining is a static scheduling technique that aims to increase a processor's CPI by pipelining more independent operations in the same loop. This technique improves upon standard loop unrolling by overlapping wind-up and wind-down overheads, effectively increasing the number of instruction executing at the same time.

More on correlating branch predictors

In general, (m, n) correlating branch predictors record the last m branches to choose from 2^m branch history tables, each of which is an n-bit predictor.

GPUs

A "standard" GPU pipeline can be summarised in five stages:

1. **Host interface**: the GPU receives commands from the CPU and pulls geometry information from the system's memory
2. **Vertex processing**: this stage receives vertices from the host interface in world coordinates and outputs them in screen coordinates
3. **Triangle setup**: in this stage, geometry is rasterised and, thanks to algorithms like backface culling, it is decided what sides of objects are actually visible on the screen
4. **Fragment processing**: the final color and lighting for every fragment is computed here, including texture mapping and other transforms. This is the typical bottleneck in modern applications
5. **Memory interface**: all the information computed until now is written to the frame buffer in order to be displayed to the user

Vertex, fragment and triangle processing are all programmable, which allows for fully customisable geometry.

The main problems with GPU programming are related to its synchronisation with the CPU: if the GPU is not able to consume all instructions coming from the CPU before its command buffer fills up, we encounter a GPU bottleneck.

Furthermore, there is no "full" synchronisation between GPU and CPU: when the CPU calls for a function to execute on a GPU, that function will be queued, but not necessarily executed right away by the GPU. This means that some data the GPU will use when executing instructions may be overwritten if the CPU gets to do that first, essentially creating a write-after-read dependency.

Modern GPU APIs implement semaphores in order to avoid these synchronisation problem. However, in this case, if the CPU attempts to modify some GPU data, it waits for the semaphore to go green, wasting precious computation time. Because of this, another approach is used: all data that needs to be sent to the GPU is inlined with the other GPU commands in the buffer, thus avoiding different references on the same data.

This, however, still reduces performance: now, instead of passing simple memory references, we are passing entire objects, which can be megabytes in size. A better solution is to allocate the new data block and initialise that one instead, fundamentally renaming conflicting data.

One last issue that can be encountered when working with a GPU occurs when the CPU tries to read output data from the GPU. This operation needs synchronisation: the GPU must first drain its entire command buffer and the CPU must wait while this happens. This massively impacts performance, because it eliminates a great part of parallelism by not providing the GPU with enough instructions to keep its buffer filled.

Message passing in caches

Message passing is an alternative to implicit communication in multicore processors caches, which has a few advantages and disadvantages:

- It provides explicit communication instead of the simple reading and writing of shared memory
- It makes control data placement more manageable
- It increases communication overhead
- It is more complex to program than simple snooping algorithms
- It introduces the problem of how cores should receive messages: via interrupts or through polling the message passing interface

---

Console processors

PlayStation 2

The PlayStation 2 CPU is a heterogeneous CPU that houses:

- The Emotion Engine: a superscalar CPU with two vector units for more efficient vector computation
- A graphics synthesiser with 16 parallel pixel processors: the main GPU in the architecture
- A MIPS CPU to handle I/O operations

Xbox 360

The Xbox 360 is a homogeneous CPU that houses three symmetrical cores with two-way simultaneous multithreading. In order to increase graphics performance, it is fitted with a VMX128 SIMD extension for each core.

PlayStation 3

The PlayStation 3 ran on a heterogeneous processor called the *Cell*, which contained:

- A 64-bit power core
- Eight specialised co-processors based on a SIMD architecture called *Synergistic Processor Unit* for graphics